Chapter 8

Model Checking with HAL

In this chapter, HAL, an automata-based verification environment for the $\pi$-calculus is introduced. The $\pi$-models gained in the previous chapters can now be verified with HAL. The HAL system is able to interface with several model checking tools to determine whether or not certain properties hold for a given specification. However, in this chapter, we have only provided some LTS of some certain $\pi$-models. A detailed about working with HAL can be found in [9, 10].

8.1 Introduction

_History Dependent automata_ (HD-automata in short) have been proposed in [38, 39] as a new effective model for name passing calculi. Like ordinary automata, HD-automata are made out of states and labeled transitions; their peculiarity resides in the fact that states and transitions are equipped with names which are no longer dealt with as syntactic components of labels, but become explicit part of the operational model. This allows one to model explicitly name creation/deallocation and name extrusion and we know these are the distinguished mechanisms of name passing calculi.

The HD-Automata Laboratory (HAL) is an integrated tool set for the specification, verification and analysis of concurrent systems. The HAL toolkit is the component of JACK[40] which provides facilities to deal with $\pi$-calculus by exploiting HD-automata. The goal of HAL is to verify properties of mobile systems specified in the $\pi$-calculus. Exploiting HAL facilities, $\pi$-calculus specifications are translated first into
HD-automata and then in ordinary automata. Hence, the JACK bisimulation checkers can be used to verify (strong and weak) bisimilarity. Automata minimization, according to weak bisimulation is also possible. HAL supports verification of logical formulae expressing properties of the behavior of $\pi$-calculus specifications. The ACTL[41] model checker provided by JACK can be used for verifying properties of $\pi$-calculus specifications, after that the $\pi$-logic formulae expressing the properties have been translated into ACTL formulae. The complexity of the model checking algorithm depends on the construction of the state space of the $\pi$-calculus agent to be verified, which is, in the worst case, exponential in the syntactic size of the agent.

8.2 HAL compatible $\pi$-calculus

The syntax of $\pi$-calculus agents supported by HAL is shown in Figure 8.1, where we use $x$ to denote generic names and $A$ to denote agent identifiers.

```
System:       S ::= Q+
Process Definition: Q ::= A(x_1, ..., x_n) = \pi (where i \neq j \Rightarrow x_i \neq x_j) ; n>=0
Process:      \pi ::= nil                      Deadlock agent(Nil)
               | \alpha . \pi                  Prefix
               | \pi_1 | \pi_2                   Parallel composition
               | (\pi_1, ..., \pi_n)           Parallel composition
               | \pi_1 + \pi_2                 Nondeterministic composition
               | + (\pi_1, ..., \pi_n)         Nondeterministic composition
               | (x) \pi                      Restriction
               | [x=y] \pi                    Name matching
               | A(x_1, ..., x_n)              Agent identifier
               | (\pi)                         Parenthesis
Action Prefixes: \alpha ::= x?(y)                 Input
                | x!y                         Output
                | tau                         Silent
```

**Figure 8.1** The $\pi$-calculus syntax compatible with HAL

Prefixes (input, output, tau, restriction and matching) take precedence on nondeterministic composition, which in turn takes precedence on parallel composition.
8.3 HAL System Overview

In the current implementation, the HAL environment consists essentially of five modules (cf. Figure 8.2): three modules perform the translations from $\pi$-calculus agents to HD-automata ($\pi$-to-hd), from HD-automata to ordinary automata (hd-to-aut) and from $\pi$-logic formulae to ordinary ACTL formulae (pl-to-ACTL). The fourth module (hd reduce) provides routines that manipulate the HD automata. The fifth module is basically the JACK environment which works at the level of ordinary automata and performs the standard operations on them like behavioral verification and model checking.

![Diagram of HAL environment]

Figure 8.2 The logical architecture of HAL environment.

8.4 HAL Commands

HAL reads commands from the standard input. At the moment, it accepts the following commands:
**define** \(A(x_1,\ldots,x_n) = \pi\)

This command defines a \(\pi\)-calculus agent. \(A\) is the identifier that is associated to the agent. \(x_1,\ldots,x_n\) are the formal parameters and \(\pi\) is the body of the agent (cf. Figure 8.1).

**build A**

This command builds the HD-automaton for the agent corresponding to identifier \(A\). The HD-automaton is saved in file \(A.hd\).

**const x**

This command declares name \(x\) as a constant name. Constant names cannot be received by an agent as values of input transitions.

### 8.5 LTS from \(\pi\)-calculus Models

In this section, some \(\pi\)-models are used to get LTS from HAL.

#### 8.5.1 LTS of Program 5.1

HAL compatible \(\pi\)-model of Program 5.1 is as follows:

```plaintext
receive_res ! terminate . nil )

define main()=(self)(foo(self))

build main
build foo
```

The LTSs of Program 5.1 are as follows:

**Figure 8.3** LTS of Program 5.1 (a) main process, (b) foo process
Chapter 8. Model Checking with HAL

8.5.2 LTS of Program 4.2

HAL compatible π-model of Program 4.2 is as follows:

- \( \text{define } s1(self) = (\text{msg}_a) (\text{msg}_c) (self ? (\text{input}_\text{pat1}).[\text{input}_\text{pat1}=\text{msg}_a]s2(self) + \\
  \quad \text{self} ? (\text{input}_\text{pat2}).[\text{input}_\text{pat2}=\text{msg}_c]s3(self)) \)

- \( \text{define } s2(self) = (\text{msg}_x) (\text{msg}_h) (self ? (\text{input}_\text{pat1}).[\text{input}_\text{pat1}=\text{msg}_x]s3(self) + \\
  \quad \text{self} ? (\text{input}_\text{pat2}).[\text{input}_\text{pat2}=\text{msg}_h]s4(self)) \)

- \( \text{define } s3(self) = (\text{msg}_b) (\text{msg}_y) (self ? (\text{input}_\text{pat1}).[\text{input}_\text{pat1}=\text{msg}_b]s1(self) + \\
  \quad \text{self} ? (\text{input}_\text{pat2}).[\text{input}_\text{pat2}=\text{msg}_y]s2(self)) \)

- \( \text{define } s4(self) = (\text{msg}_i) (\text{self} ? (\text{input}_\text{pat}).[\text{input}_\text{pat}=\text{msg}_i]s3(self)) \)

- \( \text{define start(self)} = (\text{pid})(\text{receiver}_\text{res}) (p) (\text{send}_\text{res})(\text{msg}_a) (p ! \text{pid} .\text{nil} | s1(\text{pid}) | \\
  \quad \text{receiver}_\text{res} ? (\text{dummy}).\text{nil} | p ? (\text{State}_\text{Pid}).(\text{State}_\text{Pid} ! \text{msg}_a \\
  \quad \text{.nil} | \text{send}_\text{res} ! \text{msg}_a .\text{nil} ) ) \)

- \( \text{define main()} = (\text{self}) \text{(start(self)}):^\wedge \)

build s1
build s2
build s3
build s4
build start
build main

---

Figure 8.4 LTS of Program 4.2  (a) start process, (b) s4 process (c) s1/s2/s3 process
8.5.3 LTS of Program 5.3

HAL compatible π-model of Program 5.3 is as follows:

\[
\text{define ping}(\text{self}) = (\text{po_pid})(\text{p})(\text{pong_res})(\text{pong_send_res})(\text{ping_res})(\text{pong})(\text{ping}) \ (\ p \ \! \ \text{po_pid.nil} \ | \ \text{pong}(\text{po_pid}) \ | \ \text{pong_res} \ ?(\text{dummy}).\text{nil} \ | ?(\text{Pong_ID}).\text{(Pong_ID!self. Pong_ID!ping.nil} \ | \ \text{pong_send_res} \ ?(\text{dummy}).\text{(self} \ ?(\text{input_pat}).[\text{input_pat}=\text{pong}]\text{ping_res} \ ! \ \text{pong.nil}) ))
\]

\[
\text{define pong}(\text{self}) = (\text{ping})(\text{pong})(\text{pong_res})(\text{self}?(\text{Ping_ID}).\text{self?(input_pat}).\text{[input_pat}=\text{pong}]\text{(Ping_ID!pong.nil} \ | \ \text{pong_res!pong.nil})
\]

\[
\text{define main()} = (\text{self})(\text{ping}(\text{self}))
\]

build ping
build pong
build main

8.5.4 LTS of Program 5.4

HAL compatible π-model of Program 5.4 is as follows:

\[
\text{define loop}(\text{self}) = (\text{loop_send_res})(\text{Message})(\text{self}?(\text{From}).\text{self?(Message). From!Message.nil} \ | \ \text{loop_send_res!Message.nil} \ | \ \text{loop_send_res?(dummy).loop}(\text{self}))
\]

\[
\text{define start}(\text{self}) = (\text{lpid})(\text{p})(\text{loop_res})(\text{hello})(\text{start_send_res})(\ p!\text{lpid.nil} \ | \ \text{loop(lpid).loop_res?(dummy).nil} \ | \ p?(\text{Loop_Pid}).(\text{Loop_Pid!self. Loop_Pid!hello.nil} \ | \ \text{start_send_res!self.start_send_res!hello.nil}))
\]

\[
\text{define main()} = (\text{self})(\text{start}(\text{self}))
\]

Figure 8.5 LTS of Program 5.3  (a) main/ping process, (b) pong process
build loop
build start
build main

With the recursive definition of loop process, it was not possible (there were infinite states) to get LTS from HAL, therefore, we have use nil process in place of recursive process loop. The new definition of loop process is as follows:

   From!Message.nil | loop_send_res! Message.nil |
   loop_send_res?(dummy).nil)```

Figure 8.6 LTS of Program 5.4 (a) main process, (b) start process (c) loop process.